RECONFIGURABLE GENERAL PURPOSE PROCESSOR

FIELD OF THE INVENTION

The present invention relates to reconfigurable multidimensional logic fields and their operation.

BACKGROUND INFORMATION

Reconfigurable elements are designed differently depending on the application to be executed and are designed to be consistent with the application. A reconfigurable architecture is understood in the present case to refer to modules or units (VPUs) having a configurable function and/or interconnection, in particular integrated modules having a plurality of arithmetic and/or logic and/or analog and/or memory and/or internally/externally interconnected modules arranged in one or more dimensions and interconnected directly or via a bus system.

The generic type represented by these modules includes in particular systolic arrays, neural networks, multiprocessor systems, processors having multiple arithmetic units and/or logic cells and/or communicative/peripheral cells (IO), interconnection and network modules, e.g., crossbar switches as well as known modules of the FPGA, DPGA, Chameleon, VPUTER, etc. types. Reference is made in particular in this connection to the following patents and applications by the present applicant: DE 44 16 881 A1, DE 197 81 412 A1, DE 197 81 483 A1, DE 196 54 846 A1, DE 196 54 593 A1, DE 197 04 044.6 A1, DE 198 80 129 A1, DE 198 61 088 A1, DE 199 80 312 A1, PCT/DE 00/01869, DE 100 36 627 A1, DE 100 28 397 A1, DE 101 10 530 A1, DE 101 11 014 A1,

PCT/EP 00/10516, EP 01 102 674 A1, DE 198 80 128 A1, DE 101 39 170 A1, DE 198 09 640 A1, DE 199 26 538.0 A1, DE 100 50 442 A1, as well as PCT/EP 02/02398, DE 102 40 000, DE 102 02 044, DE 102 02 175, DE 101 29 237, DE 101 42 904, DE 101 35 210, EP 01 129 923, PCT/EP 02/10084, DE 102 12 622, DE 102 36 271, DE 102 12 621, EP 02 009 868, DE 102 36 272, DE 102 41 812, DE 102 36 269, DE 102 43 322, EP 02 022 692, PACT40. Reference is made to the documents below by using the applicant's internal reference notation. These are herewith incorporated to the full extent for disclosure purposes.

The aforementioned architecture is used as an example for illustration and is referred to below as a VPU. This architecture is composed of any arithmetic or logic cells (including memories) and/or memory cells and/or interconnection cells and/or communicative/peripheral (IO) cells (PAEs) which may be arranged to form a one-dimensional or multidimensional matrix (PA), which may have different cells of any design. Bus systems are also understood to be cells here. The matrix as a whole or parts thereof are assigned a configuration unit (CT, load logic) which configures the interconnection and function of the PA. The CT may be designed as a dedicated unit according to PACT05, PACT10, PACT17, for example, or as a host microprocessor according to P 44 16 881.0-53, DE 101 06 856.9; it may be assigned to the PA and/or implemented with or through such a unit.

SUMMARY

The present invention relates to a processor model for reconfigurable architectures based on the model of a traditional processor in some essential points. For better

understanding, the traditional model will be first considered in greater detail. Resources external to the processor (e.g., main memory for programs and data, etc.) are not considered here.

A processor executes a program in a process. The program includes a finite quantity of instructions (this quantity may include multiple instances of elements) as well as information regarding the order in which the instructions may follow one another. This order is determined primarily by the linear arrangement of the instructions in the program memory and the targets of jump instructions.

Instructions are usually identified by their address. As an example, Figure 1 (a) shows a program written in VAX Assembler for exponentiation.

A program may also be interpreted as oriented graphs, where the instructions form the nodes and their order is modeled as edges of the graph. This graph is shown in Figure 1 (b). The graph has a definite start node and a definite end node (not shown in the figure; indicated by the arrows). The edges may additionally be marked with transition probabilities. This information may then be used for jump prediction. The jump prediction may in turn be used for preloading configurations into the memory of the CT of a VPU (see patent application PACT10, the full content of which has been included for disclosure purposes) and/or for preloading configurations into the configuration stack of the PAE (according to patent applications PACT13, PACT17, PACT31, the full content of which is included for disclosure purposes). By preloading configurations into the local memory of the CT (see PACT10, 17) and/or into the PAE's

local configuration cache (PACT17, 31), the configurations may then be called more rapidly as needed, which yields a great increase in efficiency.

The linear arrangement of the instructions in the memory results in more dependences than absolutely necessary; e.g., in the example shown here, instructions DECL and MULL2 are mutually independent. This is not indicated by the graph in Figure 1 (b). The model may be expanded accordingly by division nodes and combination nodes, as illustrated in Figure 1 (c).

Processors today implement such possibilities of parallel execution in hardware to some extent and distribute the operations among various arithmetic logic units. The model from Figure 1 (b) will be used for further consideration. The discussion of the additional complexity of division and combining will be shifted to a later point in time. A process also needs other resources in addition to the program for its execution. Within the processor, these include the registers and the status flags.

These resources are used to convey information between the individual program instructions. The task of the operating system is to ensure that the resources needed for execution of a process are available to it and are released again when the process is terminated. Processors today usually have only one set of registers, so that only one process may run on the processor at a time. It is possible for the instructions of two different processes to be executable in any order as long as both processes use disjunct resources (e.g., if process 1 is using registers 0-3 and process 2 is using registers 4-7).

Instructions of a processor usually have the following properties:

- An instruction is not interrupted during execution.
- The execution time for all instructions does not exceed a certain maximum value.
- Invalid instructions are recognized by the processor.

An object of the present invention is to provide a novel approach for commercial use.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1a shows a program written in VAX assembler.

Figure 1b shows the program interpreted as a graph.

Figure 1c shows an expanded model.

Figure 2 shows a subprogram in graphic representation.

Figure 3 shows an inserted subprogram call.

Figure 4a shows the position of a power at the beginning of a CIW.

Figure 4b shows how the pointer position of a register may appear at the end of a CIW.

Figure 5a shows a register before a write access.

Figure 5b shows that existing data may be deleted in such a was that a write access begins with an empty vector.

Figure 5c shows the write data may be appended to the

existing content.

Figure 5d and 5e show the state of the register after successful write operations.

Figures 6a-6e show read/write accesses.

Figure 7 shows an example of a FIFO stage.

Figure 8 shows the connection of multiple stages

Figure 9 shows a possible cache content during operation.

Figure 10a shows the free list as completely full.

Figure 10b shows memory parts affected.

Figure 11a shows a state prior to deletion.

Figure 11b shows a state after deletion.

DETAILED DESCRIPTION OF EXAMPLE EMBODIMENTS

2. Transfer of the model to the VPU architecture

An exemplary VPU architecture is a reconfigurable processor architecture as described in, for example, PACT01, 02, 03, 04, 05, 07, 08, 09, 10, 13, 17, 22, 23, 24, 31. As mentioned above, the full content of these documents is herewith incorporated for disclosure purposes. Reference is also made to PACT11, 20 and 27, which describe corresponding high-level language compilers, as well as a PACT21 which

describes a corresponding debugger. The full content of these documents is also included here for disclosure purposes.

The traditional instruction is replaced by a configuration in the known sense, referred to in the following discussion as a complex instruction word (CIW). The edges of graphs in Figure 1 (b) are formed by trigger signals to the CT. It is thus possible to implement a complete program by having the CT and/or the configuration cache of the PAEs load the following CIW after successful processing of one CIW (see PACT31 and/or as described below).

It was recognized first how a correspondence of registers of traditional processors could be implemented on the VPU architecture. It was discovered that an essential prerequisite for register implementation is based on the following:

- Since the VPU operates essentially on data streams, a register must be capable of storing a data stream and/or parts thereof.
- A register must be capable of being allocated and released. It must remain occupied as long as the program is running on the VPU (HW support of the resource management of the operating system).
- Simultaneous reading and writing (read-modify-write) of the same register should be possible.

It is explained how this may be achieved in a processor and the use of suitably modified RAM PAEs is also proposed according to the present invention. These should first be used as registers.

A detailed description of the register PAEs preferably by expanded and/or modified RAM PAEs is given in section 4 below. A configuration (CIW) is removed from the array at the moment when it requests the next CIW from the CT via a trigger. The reconfig trigger (see PACT08) may be generated either via the reconfig port of an ALU PAE or implicitly by the CT. In optimally designed versions, this should fundamentally take place from the CT.

Just as instructions on a traditional processor are not interrupted, a CIW preferably also runs on the VPU without interruption until it requests the next CIW via a trigger to the CT. It is not terminated prematurely. To be able to nevertheless ensure a regular change of instructions (which will be needed later for multitasking), the maximum execution time of a CIW has an upper limit. The second property of an instruction is thus required. It is preferably the function of the compiler to ensure that each CIW generated meets this condition. A CIW that violates this condition is an invalid instruction. It may be recognized by the hardware during execution, e.g., via a watchdog timer, which generates a trigger more or less as a warning signal after a certain amount of time has elapsed.

The warning signal is preferably managed as a TRAP by the hardware and/or the operating system. The signal is also preferably sent to the CT. An invalid CIW is preferably terminated via a reconfig trigger, which causes a reset-like deletion of all configurations in the PA and/or an exception is also preferably sent to the operating system.

Since CIWs are very long, the instruction fetch time (time between the reconfiguration trigger of the PAEs to the CT

(see PACT08) and configuration is loaded in the FILMO cache) and instruction decode time (distribution of the configuration data from the FILMO cache (see PACT10) into the configuration registers of the PAEs) are also very long. Therefore, utilization of the execution units (i.e., the PA in the VPU processor model) by a process is not very high. How this problem may be solved with multiple processors is described in section 6 below.

3. Subprograms

A subprogram in the graphic representation is a partial graph of a program having uniquely defined input nodes. The edge of the subprogram call within the graph is thus statically known. The continuing edge at the output node of the subprogram, however, is not statically known. This is shown in Figure 2. The edges of the main program (0201/0202) to the subprogram (0205) are present, but the continuation (0206) after the subprogram is not known to subprogram 0205. The particular continuation is fixedly connected to the subprogram call (indicated by dashed lines and dotted lines). It must be inserted in a suitable manner into the graphs before reaching the input node (0207, 0208). This is illustrated in Figure 3.

In traditional processors, this is usually accomplished by storing the address of the instruction following the subprogram (this is precisely the missing edge) in a call stack when the subprogram is called (call, 0203, 0204). The address may be called from there by a return.

A stack PAE is thus needed when this principle is applied to the VPU. Like register PAEs, this is a process resource and

is managed as such. The CIW, which causes the subprogram call when terminated, configures the return edge on the stack PAE. Through a trigger, the last CIW of the subprogram causes the stack PAE to remove the top entry from the stack and send it as a reconfiguration call to the CT.

In implementing a stack, one of the following methods may be used, for example:

- An implementation within the CT. The stack is implemented in the software or as a dedicated hardware unit within the CT. A special config ID (e.g., -1) may be reserved as the return. When the CT receives this ID, it replaces it by the top entry of its locally managed stack.

- A stack PAE, which may be designed as a modified RAM PAE according to PACT13 (Figure 2), for example. Stack overflow and stack underflow are exceptions which are preferably forwarded to the operating system.

4. The Register PAE

A traditional processor register contains a data word at each point in time. An instruction is able to read, write or modify the contents of the register (read-modify-write).

A VPU register will have the same properties, but instead of a single value, according to the present invention it will contain a value vector or parts thereof. It is possible and usually preferable for a VPU register to be organized as a type of FIFO. In certain cases, however, random access may also be necessary. The three types of register access mentioned above are explained in detail below. Random access

is not discussed here.

Read access. At the start of a CIW, the register contains a data vector of unknown length. The individual elements of the vector are removed sequentially. With the last element of the vector, a trigger is generated, indicating that the register is now empty and the CIW may terminate. The status of the register may be characterized using three pointers which point to the first entry (0403) in the data vector, the last entry (0401) and the current entry (0402). The position of the pointer at the beginning of a CIW is shown as an example in Figure 4 (a), where the pointer for the current entry points at the first entry.

Figure 4 (b) shows in a first example how the pointer position of a register may appear at the end of a CIW. The vector has not been read completely in the case illustrated here.

Consequently, a decision must be made regarding what happens with the register contents. There are preferably the following options:

- The register is emptied. All unprocessed data is deleted. The pointer for the current entry points at the last entry.
- The register is reset at the original state. The next CIW may thus again access the full data vector. The pointer for the current entry is reset to point at the first entry.
- Only the data already read is removed from the register. The unread data is then available for the next CIW. The pointers are not modified. Subsequently, the values between the first entry and the current entry are removed from the register. They are then no longer available for further operations.

The third option is of interest in particular when a CIW is unable to completely process the data vector because of the maximum execution time for a CIW. See also section 7.

Write access. Data here is written sequentially into the register. A trigger is generated when the filling status of the register has reached a certain level. Depending on the CIW, this may be one of the following preferred possibilities:

- The register is completely full.
- There are still precisely n entries in the vector that are free. This takes into account the latency time in the CIW through which n values after the trigger are still running to the register.
- The register is m% full.

A CIW which attempts to write into a completely full register is invalid and is terminated with an exception (illegal opcode). At the start of the CIW, the status of the register should be determined. Figure 5 (a) shows a register before a write access which still contains data. Existing data may be deleted in such a way that the write access begins with an empty vector (Figure 5 (b)). As an alternative, the write data may also be appended to the existing content. This is shown in Figure 5 (c). This case is of interest when the preceding CIW was unable to generate the complete vector because of the maximum execution time.

Figures 5 (d) and (e) show the state of the register after successful write operations. The newly written data is indicated here with hatching.

Simultaneous read/write access. The restriction to pure read

access or write access requires a greater number of registers than necessary. When data is removed from a register by read access, this yields locations which may be occupied by write data. It is only necessary to ensure that write data cannot be read again by the same CIW, i.e., there is a clear separation between the read data of a CIW and the write data of the CIW. For this purpose a virtual dividing line (0601) is introduced into the FIFO. The register has been read completely when this dividing line reaches the output of the FIFO. Suitable means may be implemented for defining this virtual dividing line.

If a write access for a data word is not executable because the register is still blocked by unread read data, the CIW is terminated and an illegal opcode exception is generated. The behavior of the register is otherwise exactly the same as that described for read and write access. In addition, one should specify what is to happen with the virtual dividing line between the read data and write data. This dividing line may remain at the location it is in at the moment. This is beneficial if a CIW must be terminated because of the time restriction. As an alternative, the dividing line may be set at the end of all data.

Combined read/write accesses are problematical, however, if the CIW has been terminated with an exception. In this case, it is no longer readily possible to reset the registers to their values at the start of the CIW. Debugging may then be hindered at the least (see also the following discussion in section 8).

Figure 6 illustrates the functioning using an example, where the virtual dividing line is labeled 0601. At the beginning,

the register contains data (a) which is subsequently read partially (b) or completely (c). Newly written and read entries are indicated here by different types of hatching. Figures 6 (d) and (e) show the state of the register after the required pointer update, which alters the position of the dividing line. This is not an explicit step, but is shown here only for the purpose of illustration. The entries that have been read must be removed immediately to make room for the new entries to be written.

A process, i.e., a program which shares resources with other programs in a multitasking operation in particular, must allocate each required register before it may be used. This is preferably accomplished by an additional configuration register within the RAM PAE and/or the register PAE, an entry also being made indicating to which process the register now belongs. This configuration is retained over reconfigurations. The register must be explicitly enabled by the CT. This happens on termination of a process, for example. With the configuration of each CIW, the registers must be notified of which process the CIW belongs to. This makes it possible to switch between multiple register sets. This process is described in greater detail in section 6 below.

5. Interrupts

A distinction is made between two different types of interrupts. First there are hardware interrupts, where the processor must respond to an external event. These are usually processed by the operating system and are not visible for the ongoing processes. They are not discussed further here. The second type is the software interrupts

which are frequently used to implement asynchronous interactions between the process and the operating system. For example, it is possible under VMS to send a read request to the operating system without waiting for the actual data. As soon as the data is present, the operating system interrupts the running program and calls a procedure of the program asynchronously. This method is known as an asynchronous system trap (AST).

This method may also be used in the same way on the VPU. To do so, support may be provided in the CT. The CT knows whether an asynchronous routine must be called up for a process. In this case, the next request coming from the array is not processed directly but instead is stored.

Instead, a sequence of CIWs is inserted, which first saves the processor status (the register contents), which executes the asynchronous routine and which then restores the register content. The original request may be subsequently processed.

6. Multitasking

As described above in section 2, the VPU architecture may, under some circumstances, not be optimally utilized with only one process because very long loading and decoding times occur, e.g., due to the length of the CIWs. This problem may be solved by simultaneous execution of multiple processes. According to the present invention, several register sets are provided on the VPU for this purpose, making it possible to simply switch between register sets when changing context without requiring any complex register clearance and loading operations. This also makes it

possible to increase the processing speed.

During execution of CIWs of the processes, enough time is available to retrieve the instructions of the current process and distribute them via the FILMO to the PAEs and/or to load them from the configuration cache into the PAEs (see PACT31). The optimum number of register sets may be determined as a function of the average execution time of a CIW and the average loading and decoding times of the CIWs.

The latency time may be compensated by a larger number of register sets. It is important for the functioning of the method that the average CIW running time is greater than the amount of time effectively needed for loading and/or decoding the CIW in each case.

The corresponding registers of the different register sets are then at the same PAE address for the programmer. In other words, at any point in time, only the registers of one register set may be used. The change in context between the register sets may be implemented by transmitting the corresponding context to the PAEs before each CIW. The context switch may take place automatically as depicted in detail by the PUSH/POP operations according to PACT11 and/or by a special RAM/register PAE hardware, as depicted in PACT13 Figure 21. Both cases involve a similar stack design in the memory. Each stack entry stores the data of a process. A stack entry includes the complete content of all registers, in other words, all memory cells of all memories which function as registers for a process. Likewise, according to PACT11, a stack entry may also contain PA-internal data and states.

In general, more processes will be present on a system than there are register sets in the processor. This means that a process must occasionally be removed from the processor. To do so, as in the case of the software interrupt, an edge of the program graph is divided by the CT. The register contents of the process are saved and the processor resources i (registers, stack PAEs, etc.) allocated by the process are freed again. The resources thereby freed are then allocated by another process. The register contents stored for this process are then written back again and the process is continued on this divided edge. The register contents may then be saved and reloaded via CIWs.

7. CIW and Loops

On the basis of the property required above, namely that a CIW must terminate after a certain maximum number of cycles at the latest, general loops may not be translated directly into a CIW. It is always possible to translate the loop body into a CIW and to execute the loop control via reconfiguration. However, this often means a considerable sacrifice in terms of performance. This section shows how a loop may be reshaped to minimize the number of reconfigurations.

The following program fragment is assumed below:

```
while (condition) {
something;
}
```

the running time of "condition" should be determined as "something" or it should be possible to make an upper

estimate. The loop may then be reformulated as follows:

```
while (1) {
if (!condition) goto finish;
something;
}
finish:
```

The body of the loop may now be iterated as often as allowed by the maximum running time of the CIW. A new variable z is introduced for this purpose; this variable does not occur either in "condition" or in "something." The program now looks as follows:

```
while (1) {
for (z=0; z<MAX; z++) {
if (!condition) goto finish;
something;
)
)
finish:
```

The "for" loop has a maximum running time which may be determined by the compiler. It may therefore be mapped onto a CIW. MAX is determined by the compiler as a function of the maximum running time and the individual running times of the instructions.

The resulting CIW has two output edges. The output via goto leads to the next CIW; the output via the regular end of "for" forms an edge on itself. The endless loop is implemented via this edge.

8. Debugging

In the traditional processor, debugging is performed on an instruction basis, i.e., the sequence of a program may be interrupted at any time between two instructions. At these interruption points, the programmer has access to the registers, may look at them and modify them. Interruption points may be implemented in various ways. First the program may be modified, i.e., the instruction before which the interruption is to occur, is replaced by other instructions which call the debugger. In the graphic model, this corresponds to replacing one node with another node or with a partial graph. Another method is based on additional hardware support, where the processor is notified of which instruction the program is to be interrupted at. The corresponding instruction is usually identified by its address.

Both possibilities may be applied to the VPU according to the present invention. One CIW may be replaced by another CIW, by action of the debugger, for example. This CIW may then, for example, copy the register contents into the main memory, where they may either be analyzed using a debugger external to the VPU or alternatively the debugger may also run on the VPU.

In addition, hardware support, which identifies CIWs on the basis of their ID when requested and then calls up the debugger, may also be provided in the CT. In addition, an interruption may also be fixedly attached to an edge of the graph because interruptions are present explicitly in contrast with traditional program code.

The type of debugging described above is completely adequate for traditional processors because the instructions are usually very simple. There is a sufficiently fine resolution of the observable points. In addition, the programmer may rely on the individual instructions being correct (usually ensured by the processor manufacturer).

On the VPU, however, it is possible for the programmer to define the CIWs which form a type of "processor instructions." Accordingly, instructions defined in themselves in this way may be defective. Debugging of the individual instructions is thus preferably designed in the manner referred to below as microcode debugging. Microcode debugging is designed in such a way that the programmer gains access to all internal registers and data paths of the processor. It has been recognized that the complexity necessary for this is readily justified by the increased functionality.

Hardware support for this is possible but very complex and is not appropriate for pure debugging purposes. Therefore, as an alternative, the status of the processor before the instruction in question is saved and the actual instruction is executed on a software simulator. This is the preferred method of debugging VPUs according to PACT11. The data and states are preferably transferred to the debugger via a bus interface, memory and/or preferably via a debugging interface such as JTAG. A debugger according to PACT21 is preferably used, preferably containing a mixed-mode debugger having an integrated simulator for processing the micro debugging.

In a suitable programming model, the debugger may also be

called when an exception occurs within an instruction. It is appropriate here that the registers may be reset back to the state before the start of the instruction and that no other side effects have occurred. Then the instruction in question may be started in the software simulator and simulated until up to the occurrence of the exception.

Particularly preferred debugging mechanisms are described in detail in PACT21.

Microcode debugging may preferably be implemented by configuring a debugging CIW before or after processing a CIW. This debugging CIW first receives all the states (e.g., in the PAEs) and then writes them into an external memory through a suitable configuration of the interconnection resources. The PUSH/POP methods described in PACT11 may be used here particularly preferably. This may preferably take place via an industry standard interface such as JTAG. Then a debugger may receive the data from the memory or via the JTAG interface and, if necessary, simulate it further incrementally in conjunction with a simulator (see PACT21), thus permitting microcode debugging.

9. Distributed configuration cache

On the basis of the central configuration cache in FILMO, it takes a relatively long time when using such a cache, which is not obligatory, until a configuration is distributed to the individual PAEs of a PAC. This section will now describe a preferred method for shortening this period of time. A similar alternative or additional method is also already described in PACT31, the full content of which is herewith incorporated for disclosure purposes.

For this purpose, each PAE has its own local cache which stores the configuration data of various configurations for precisely this PAE. The fact that a PAE has not received any data from a configuration is also stored. For each configuration requested, the cache may thus make one of the following statements:

- The configuration data is present in the cache.
- No data is needed for this configuration.
- Nothing is known about this configuration.
- Configuration data is needed but it is not available in the cache (e.g., due to the length of the configuration, RAM preload, etc.).

The last two statements may be combined here. With both statements, the code or the fact that no code is needed must be requested. An order for a configuration is sent by the FILMO as a broadcast on the test bus to all PAEs. If all PAEs have the configuration in their local cache, it may be started via broadcast on the config bus. In the ideal case, the start of the configuration thus requires the transmission of only a single configuration word.

If a PAE does not have the configuration data, this fact is reported back to the FILMO. In the simplest case, this is done via a reject on the existing line. The FILMO then knows on the basis of this signal that at least one PAE of the PAC does not have the configuration data. It may then transmit the complete data. As an alternative, each PAE may trigger separately a request for the data. In this case a suitable compromise must be made between the number of requests and the quantity of configuration data to be transmitted. Small PAC sizes are advantageous here because of the lower latency on the configuration bus.

Design of the cache

A cache is generally always composed of two parts. One part contains the actual data (here the configuration words, 0902) while the other part contains management information (here the configuration numbers contained as well as their age, 0901).

First the management part is described.

It is desirable for the configuration which has not been used for the longest period of time to be removed from the cache if this is necessary. As long as only new configurations are requested, the entries in the FIFO are sorted correctly. If a configuration is requested for which there is already an entry in the FIFO, this entry must be removed from the FIFO. It is then reinserted again at the end. Figure 7 shows an example of a FIFO stage modified for this purpose. The modules shown with hatching are in addition to a normal FIFO stage according to the related art. They compare via the comparator (0701) the configuration number of the data content of the stage with the requested configuration number and, if they are the same, generate an ack (0702) for that stage. Thus, the data of the stage is read via the multiplexer (0703) and all the other values move up by one stage. The entries in this FIFO also contain additional information in addition to the configuration number. This is either a pointer (address) to the configuration data or one of the two possibilities "no data necessary" (e.g., coded as 0) or "data must be requested," (e.g., -1). Figure 8 shows the connection of multiple stages, where the read chain is initialized with the required configuration number and the status -1. This

value then comes out unchanged at the output of the read chain exactly when the configuration number is not stored in the FIFO. The output of the read chain may thus be used in any case to write the configuration number into the FIFO. Signal ack_in is activated when the FIFO is full and the desired configuration number is not in the FIFO. This is the only case when the oldest entry must be removed from the FIFO because the management memory is full. The actual data memory is organized as a chained list because of the different number of configuration words per configuration. Other implementations are also conceivable. A chained list may then be implemented easily as a RAM by storing the address of the following data word in addition to the data.

In addition to the lists for the actual configurations, a free list is carried, listing all the entries which are not being used. This must be initialized first after a reset.

Figure 9 shows a possible cache content during operation. Free entries in the data memory are white, while entries occupied by a configuration are shown with hatching. Configurations need not be located at successive addresses. Configuration 18 has no configuration data and therefore does not also have a pointer in the data memory.

A new configuration is written into the free list in the data memory. In doing so the pointer information of the data memory is not modified. Only for the last data word of a configuration is the pointer information altered to indicate that the list is now being modified here. The pointer to the free list points at the next entry.

It may happen that the space in the free list is not

sufficient to completely accommodate the incoming configuration data. In this case, a decision must be made as to whether an old configuration is to be removed from the data memory or whether the current configuration is not to be included in the cache. In the latter case, the subsequent configuration words are discarded. Since no pointer has been modified, the free list remains the same as before and only a few unused data words have a different value. The decision as to which configuration should no longer be in the cache (the oldest or the current) may be made on the basis of the number of configuration words already written. There is little point in removing several cached configurations to make room for a long RAM initialization, for example.

If the oldest configuration is to be removed, it is removed from the FIFO. The pointer for the last entry in the free list is set at the value taken from the FIFO. After this address, configuration may be continued in the accustomed manner.

Figure 10 shows an example of this. Configuration no. 7 is to be reconfigured. Figure 10 (a) shows the free list as completely full. A decision is made to remove the oldest configuration (no. 5) from the cache and to write configuration no. 7 into the cache. To do so, the pointer is moved from the end of the free list to the start of former configuration 5. The free list is thus lengthened again and space is again available for new configuration words. The memory parts affected in this step are shown with contradiagonal hatching in Figure 10 (b). With a suitable division of the memory, this may take place in one cycle. With the last configuration word, the corresponding pointer points at the end and the free pointer points at the next

entry. Space in the data memory is then not only freed up again when needed by the inclusion of a new configuration, but also if the management memory is full and therefore an entry is removed from the management memory, the free list in the data memory must be adapted. To do so, either the pointer at the end of the free list or at the end of the configuration being freed up is adapted. Both types of information are not yet available at this point. It is now possible to move through one of the lists until reaching the end. However, this is time-consuming. As an alternative, an additional pointer to the particular end of a configuration is stored in the management memory. Modification is then easily possible. The free pointer receives the starting address of the old configuration, and the pointer at the last configuration word in the data memory points at the free pointer.

This is illustrated in Figure 11. The pointers to the configuration ends are shown with dashed lines. Figure 11 (a) illustrates the situation before deletion, Figure 11 (b) illustrates the situation afterwards.

10. Optimization of bus allocation

The buses are currently defined explicitly by the router. This may result in two configurations overlapping on a bus and therefore not being able to run simultaneously although on the whole enough buses would be available.

It has been recognized that it does not matter in terms of the algorithm which bus carries a connection. Therefore, it is proposed that bus allocation be performed dynamically by the hardware and the hardware be provided with a suitable

dynamic bus allocator. A configuration specifies only that it needs a connection from point A to point B within a row. An arbiter in the hardware which is able to work per row either via proximity relationships in a distributed manner or at a central location for the row then selects which of the available buses is in fact used. In addition, buses may be dynamically rearranged. Two short non-overlapping buses which have been configured to different bus numbers on the basis of a previous allocation may be rearranged to the same bus number when resources become available. This creates space for longer connections in the future.